

2

NAVAL POSTGRADUATE SCHOOL

Monterey, California

AD-A235 448



DTIC
ELECTE
MAY 02 1991
S B D

**CLUSTERING, CONCURRENCY CONTROL, CRASH
RECOVERY, GARBAGE COLLECTION, and SECURITY in
OBJECT-ORIENTED DATABASE MANAGEMENT
SYSTEMS**

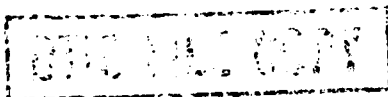
Everton G. de Paula, Captain, Brazilian AF
Michael L. Nelson, Major, USAF

February 1991

Approved for public release; distribution is unlimited.

Prepared for:

Naval Postgraduate School
Department of Computer Science, Code CS
Monterey, California 93943-5100



91 F 01 032


NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral R. W. West, Jr.
Superintendent

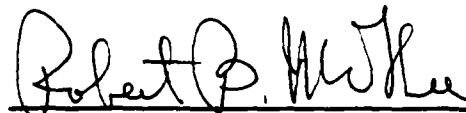
Harrison Shull
Provost

This report was prepared in conjunction with research funded by the Naval Postgraduate School Research Council.

Reproduction of all or part of this report is authorized.


MICHAEL L. NELSON
Assistant Professor
of Computer Science

Reviewed by:


ROBERT B. MCGHEE
Chairman
Department of Computer Science

Released by:


PAUL J. MARTO
Dean of Research

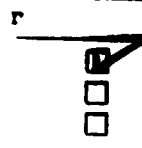
UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

| REPORT DOCUMENTATION PAGE | | | | Form Approved OMB No. 0704-0188 | |
|--|-------|--|---|--|----------------------------|
| 1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED | | | 1b. RESTRICTIVE MARKINGS | | |
| 2a. SECURITY CLASSIFICATION AUTHORITY | | | 3. DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE: DISTRIBUTION IS UNLIMITED. | | |
| 2b. DECLASSIFICATION/DOWNGRADING SCHEDULE | | | | | |
| 4. PERFORMING ORGANIZATION REPORT NUMBER(S) NPSCS-91-008 | | | 5. MONITORING ORGANIZATION REPORT NUMBER(S) | | |
| 6a. NAME OF PERFORMING ORGANIZATION Dept. of Computer Science Naval Postgraduate School | | 6b. OFFICE SYMBOL (If applicable) CS | 7a. NAME OF MONITORING ORGANIZATION | | |
| 6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943 | | | 7b. ADDRESS (City, State, and ZIP Code) | | |
| 8a. NAME OF FUNDING/SPONSORING ORGANIZATION Naval Postgraduate School | | 8b. OFFICE SYMBOL (If applicable) CS | 9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER OM&N Direct Funding | | |
| 8c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943 | | | 10. SOURCE OF FUNDING NUMBERS | | |
| | | | PROGRAM ELEMENT NO. | PROJECT NO. | TASK NO. |
| 11. TITLE (Include Security Classification) Clustering, Concurrency Control, Crash Recovery, Garbage Collection, and Security in Object-Oriented Database Management Systems (UNCLASSIFIED) | | | | | |
| 12. PERSONAL AUTHOR(S) Everton G. de Paula and Michael L. Nelson | | | | | |
| 13a. TYPE OF REPORT Summary | | 13b. TIME COVERED FROM _____ TO _____ | | 14. DATE OF REPORT (Year, Month, Day) 91 Feb 25 | |
| 15. PAGE COUNT 19 | | | | | |
| 16. SUPPLEMENTARY NOTATION | | | | | |
| 17. COSATI CODES | | | 18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) Object-Oriented Database Management Systems, Clustering, Concurrency Control, Crash Recovery, Garbage Collection, Security | | |
| FIELD | GROUP | SUB-GROUP | | | |
| | | | | | |
| | | | | | |
| 19. ABSTRACT (Continue on reverse if necessary and identify by block number) This paper presents considerations about several topics that have a direct influence on data reliability and performance in object-oriented database management systems. These topics are: physical storage management (clustering), concurrency control, crash recovery, garbage collection, and database security. Each topic is illustrated by its application to the Tactical Database as designed for the Low Cost Combat Direction System. | | | | | |
| 20. DISTRIBUTION/AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS | | | 21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED | | |
| 22a. NAME OF RESPONSIBLE INDIVIDUAL Michael L. Nelson | | | 22b. TELEPHONE (Include Area Code) (408) 646-2449 | | 22c. OFFICE SYMBOL CSNe |

TABLE OF CONTENTS

| | | |
|-------|--|----|
| 1 | INTRODUCTION | 1 |
| 2 | OBJECT CLUSTERING | 1 |
| 2.1 | ANALYSIS OF THE OODBMS CLUSTERING FACILITIES | 2 |
| 2.2 | STORING INSTANCES OF THE SAME CLASS IN SEPARATE CLUSTERS | 2 |
| 2.3 | CLUSTERING INSTANCES OF A CLASS TOGETHER | 3 |
| 2.4 | CLUSTERING INSTANCES OF A COMPONENT CLASS | 3 |
| 2.5 | CLUSTERING COMPONENTS OF A COMPOSITE CLASS | 3 |
| 2.6 | CLUSTERING ABSTRACT CLASSES | 3 |
| 2.7 | ADDITIONAL CONSIDERATIONS FOR CLUSTERING | 3 |
| 2.8 | CLUSTERING IN THE TACTICAL DATABASE | 4 |
| 2.8.1 | CLUSTERING OF COMPOSITE CLASSES | 4 |
| 2.8.2 | CLUSTERING OF CLASSES THAT ARE USED TOGETHER | 4 |
| 2.8.3 | CLUSTERING OF THE OTHER CLASSES | 5 |
| 3 | CONCURRENCY CONTROL | 5 |
| 3.1 | CONCURRENCY CONTROL MECHANISMS | 6 |
| 3.1.1 | OPTIMISTIC CONCURRENCY CONTROL | 6 |
| 3.1.2 | PESSIMISTIC CONCURRENCY CONTROL | 6 |
| 3.1.3 | OPTIMISTIC vs PESSIMISTIC CONCURRENCY CONTROL | 7 |
| 3.2 | OODBMS CONCURRENCY CONTROL | 8 |
| 3.3 | CONCURRENCY CONTROL IN THE TACTICAL DATABASE | 9 |
| 4 | CRASH RECOVERY | 10 |
| 4.1 | OODBMS RECOVERY | 10 |
| 4.2 | RECOVERY IN THE TACTICAL DATABASE | 10 |
| 5 | GARBAGE COLLECTION | 10 |
| 5.1 | GARBAGE COLLECTION IN THE TACTICAL DATABASE | 11 |
| 6 | DATABASE SECURITY | 11 |
| 6.1 | OODBMS SECURITY | 12 |
| 6.1.1 | LOGIN AUTHORIZATION | 12 |
| 6.1.2 | NAME HIDING | 12 |
| 6.1.3 | PROCEDURAL PROTECTION | 13 |
| 6.1.4 | NONPROCEDURAL PROTECTION | 13 |
| 6.2 | SECURITY IN THE TACTICAL DATABASE | 13 |
| 7 | CONCLUSIONS | 13 |
| | REFERENCES | 14 |
| | INITIAL DISTRIBUTION LIST | 15 |



A-1

1 INTRODUCTION

Object-oriented database management systems (OODBMS) are still a relatively new area, with many unanswered questions as to their use and performance. In this paper (which we call "OODBMS Topics", for short), we present information on several issues pertaining specifically to OODBMSs: clustering (how to organize the objects on disk), concurrency control (how to manage simultaneous use of the objects by various users), crash recovery (from both soft (software) and hard (hardware) crashes), garbage collection (how/when to reorganize main memory as objects move between main memory and disk), and security (how to control access to the various objects stored in the OODBMS).

Although this might seem to make for a rather lengthy report of several unrelated subjects, each section is actually fairly short. And, to effectively manage an OODBMS, all of these topics must be considered. Though all examples given herein are from the Tactical Database (TDB) of the Low Cost Combat Direction System (LCCDS) [dePa90, SS90], we have generalized the concepts so that they can be applied to virtually any OODBMS application.¹ Also, since preliminary studies have indicated that the GemStone OODBMS [BMOP89, Serv89] is the system of choice for the LCCDS [Ross89], special emphasis will be given to this particular system.

2 OBJECT CLUSTERING

The way in which objects are stored on disk affects the overall performance of the system. In general, when objects which are often used together are clustered together on disk (i.e., placed in contiguous storage areas), overall retrieval time is reduced.

In order to determine how the objects should be clustered on disk, the following guidelines are proposed:

- Analyze the clustering facilities provided by the OODBMS being used.

- Determine whether or not instances of the same class can be stored in separate clusters.

- Place all instances of a class that is not a component of several composite classes in the same cluster, if possible.

- Do not cluster together instances of a class which is used as a component of several composite classes, if possible.

- Place all components of a composite class in the same cluster, if possible.

- Cluster only concrete classes.

The guidelines for object clustering proposed in this paper are based on the principle that objects which are often used together should be clustered together on disk. However, it should be remembered that the ideal clustering scheme will vary depending on the type of queries most often posed to the database.

¹It should be realized that while the LCCDS TDB is used for examples, some of the issues involved have been simplified; the purpose of this paper is to present various OODBMS topics, not the LCCDS TDB itself. For more exact information on the LCCDS TDB, refer to [dePa90].

2.1 ANALYSIS OF THE OODBMS CLUSTERING FACILITIES

The physical storage of objects on disk depends on the actual OODBMS being used. Different systems use different default clustering mechanisms. Additionally, they may or may not allow user-defined clustering.

In ONTOS [Onto90], all elements (components) of an object are stored together so that they can be retrieved efficiently. Additionally, related objects can be clustered together on disk in any arbitrary, programmer-defined grouping. Therefore, objects may be clustered according to anticipated application usage(s).

Clustering objects together means storing them in the same segment. **Segments** are variable-sized atomic units of transfer between secondary storage and main memory [Onto88]. Whenever one of the objects is accessed from disk and brought into main memory, all objects in that cluster are also brought into main memory. Subsequent access to any other object in that segment is then a main memory access rather than a disk access. Thus, when done properly, clustering can be used to great advantage in application performance.

In ORION [BCGK87, KBCG89], all instances of a class are placed in the same segment. Thus a class is associated with a single segment and all of its instances reside therein. The user does not have to be aware of segments as ORION automatically allocates a separate segment for each class. However, for clustering composite objects it is often necessary to store instances of several classes in the same segment. The user must specify which classes should be stored together in the same segment.

One of the original goals of GemStone was that the system would provide features for managing the placement of objects on disk. The database administrator, or an application programmer, should be able to specify how certain objects should be clustered on the disk. GemStone provides two clustering methods: one basic and another that is more sophisticated.

The basic clustering method simply assigns each class to a disk page. It does not attempt to cluster the class' instance variables. If the user wishes to cluster the instance variables of an object, a special method must be defined to do so, using the basic clustering method as a tool.

The more sophisticated method does a depth-first traversal of the tree representing the instance variables of the instances of a class. That is, it writes to the disk the object's first instance variable, then the first instance variable of that variable, then the first instance variable of that variable, and so on, to the bottom of the tree. It then backs up and visits the variables which it missed before, repeating the process until the entire tree has been stored. [Serv89]

2.2 STORING INSTANCES OF THE SAME CLASS IN SEPARATE CLUSTERS

If the system permits instances of the same class to be stored in different clusters, then the user has complete freedom in deciding how to cluster them. In this case, all components of a composite object can be stored together. However, if the system requires that all instances of a class be stored together in the same cluster (i.e., if instances of a class cannot be stored in separate clusters), then the user is limited to creating only clusters of classes.

2.3 CLUSTERING INSTANCES OF A CLASS TOGETHER

Placing all instances of a class in the same cluster allows the retrieval of all the instances of that class in a single disk access. However, if a class has a large number of instances, the cluster may become too large (possibly even larger than main memory) to be efficiently read into main memory. In this case, the only solution is to store the instances in more than one cluster.

2.4 CLUSTERING INSTANCES OF COMPONENT CLASSES

When a class is a component of several composite classes, the number of classes to be stored together may become too large to be handled efficiently by the system. If the system permits, one solution to this problem is to not cluster the instances of the component class together. Instead, the clustering is done in such a way that each of the instances of the component class is stored along with its corresponding composite object.

2.5 CLUSTERING COMPONENTS OF A COMPOSITE CLASS

Ideally, all of the components of a composite object should be stored together [BCGK87]. However, if a class has several component classes, then a cluster including all of these classes might be too large to be handled efficiently by the system.

2.6 CLUSTERING ABSTRACT CLASSES

Concrete classes may have a large number of instances. Besides the space necessary for their class definitions, they may require a large amount of storage space for their instances. Abstract classes, on the other hand, have no instances. Therefore, the only storage space occupied by these classes is the space necessary to store their class definitions.

The primary purpose of clustering objects together on disk is to reduce the amount of time required to retrieve information. This information is contained (stored) in the instances of the class, not in the class definition itself. Therefore, it is not necessary to cluster abstract classes.

2.7 ADDITIONAL CONSIDERATIONS FOR CLUSTERING

Some additional considerations or options for placement of objects on disk are as follows [HZ87]:

One object per segment is intended for very large objects, since they are costly to transfer and tend to be accessed individually.

Partitioning based on property values is similar to indexing. In using properties, specific values, such as "green", or numeric intervals, such as $0 < n < 9$, may be specified. This method allows the user to separate objects containing a property value of particular interest into one segment. Notice, however, that changing an object's color from "green" to "blue" may violate the original specification of a segment whose objects were to have the color property value of "green". This might require that the object be moved to another, more appropriate segment.

These methods may also be used to tailor object placement on disk to meet expected needs [HZ87].

2.8 CLUSTERING IN THE TACTICAL DATABASE

2.8.1 CLUSTERING OF COMPOSITE CLASSES

In the TDB, the variable TIME OF POSITION, an instance of the class TIME, was defined for most of the classes. Consequently, most of the objects in the TDB are composite objects which have TIME as one of their components. In order to store all instances of TIME together with all the instances of the other classes, it would be necessary to create a cluster that would include most of the classes defined for the TDB (i.e., most of the classes would have to be clustered together on disk). The resulting cluster would certainly be much too large to be used efficiently.

Therefore, instances of the class TIME should not be clustered together. The clustering should be done in such a way that the instances of TIME (represented by the variable TIME OF POSITION) are clustered together with their corresponding composite object.

A similar situation occurs with the classes ANGLE, LATITUDE, LONGITUDE, GEOGRAPHICAL POSITION, DIRECTION, RELATIVE POSITION, CLOSE CPA, COLLISION CPA, and HMS. Each of these classes is a component class of several different composite classes. Storing them all together along with their respective composite classes would result in the creation of large and inefficient clusters. Thus, instances of these classes should not be clustered together. Rather, they should be clustered along with their corresponding composite objects.

The classes WIND and MAGNETIC VARIATION, on the other hand, are component classes only of the composite class OWNSHIP. Therefore, they should be clustered together with the class OWNSHIP. Similarly, the classes DISTANCE RANGE and TIME RANGE are components only of the classes CLOSE CPA and COLLISION CPA. Therefore, these four classes should be clustered together. Also, since the class WAYPOINT is a component class only of the class ROUTE, these two classes should be clustered together.

2.8.2 CLUSTERING OF CLASSES THAT ARE USED TOGETHER

In the TDB scenario, every new track is first classified as a tentative track and later, after a valid course and speed is established for it, becomes either an air track, a surface track, or a subsurface track. Thus, the class TENTATIVE TRACK is often used together with the classes AIR TRACK, SURFACE TRACK, and SUBSURFACE TRACK. Therefore, these four classes should be clustered together.

Additionally, all of the tracks' relative positions and closest points of approach (CPAs) are calculated based on Ownship's geographical position. Consequently, the classes TENTATIVE TRACK, AIR TRACK, SURFACE TRACK, and SUBSURFACE TRACK are generally used together with the class OWNSHIP. However, this is also true for all special points. That is, their relative positions and closest points of approach (CPAs) are also calculated based on Ownship's geographical position. Thus, the classes REFERENCE POINT, DATA LINK REFERENCE POINT, WAYPOINT, FORMATION CENTER, NAVIGATION HAZARD, MAN IN WATER, and POSITION AND INTENDED MOVEMENT are also used together with the class OWNSHIP.

Therefore, there are four main clustering possibilities: cluster OWNSHIP with all tracks; cluster OWNSHIP with all special points; cluster OWNSHIP with both tracks and special points; or do not cluster OWNSHIP with either.

Clustering OWNSHIP with both tracks and special points would result in a cluster that would probably be too large to be handled efficiently by the system, since most of the classes defined for the TDB are either subclasses of the class TRACK or subclasses of the class SPECIAL POINTS.

Although clustering OWNSHIP with all special points or not clustering OWNSHIP with either tracks or special points are viable alternatives, it is important to realize that the amount of time to retrieve information about a track is critical to the safety of the ship - it is necessary to have all information about a track available as fast as possible. The fastest way to retrieve information about a track is to cluster the class OWNSHIP with all tracks (assuming that there are no problems with memory size).

Therefore, the classes TENTATIVE TRACK, AIR TRACK, SURFACE TRACK, SUBSURFACE TRACK, and OWNSHIP should all be clustered together on disk. Remember that the classes WIND and MAGNETIC VARIATION should also be clustered together with the class OWNSHIP.

2.8.3 CLUSTERING OF THE OTHER CLASSES

The remaining concrete classes are DATA LINK REFERENCE POINT, FORMATION CENTER, MAN IN WATER, NAVIGATION HAZARD, POSITION AND INTENDED MOVEMENT, REFERENCE POINT, TDB AREA, and USERDEFINED OBJECT. The only class that is often used together with any of these classes is the class OWNSHIP. However, it was previously decided that OWNSHIP should be clustered with all the classes that represent tracks.

Thus, there is no strong reason for clustering any of these classes together. However, it is almost always practical to cluster all of the instances of a class together. In order to achieve this, the classes DATA LINK REFERENCE POINT, FORMATION CENTER, MAN IN WATER, NAVIGATION HAZARD, POSITION AND INTENDED MOVEMENT, REFERENCE POINT, TDB AREA, and USERDEFINED OBJECT should each be stored in a separate cluster.

3 CONCURRENCY CONTROL

In a database management system (DBMS) which provides concurrent access to multiple users, each user should see a consistent version of the data, regardless of how many other users are active or what they are doing. When a user logs in, the DBMS establishes a logical entity called a **session**, which is analogous to an operating system session, job, or process. A separate session is created each time a user logs in, and the DBMS monitors, serves, and protects each session independently.

In general, the DBMS prevents inconsistencies by encapsulating the operations of each session into units called **transactions**. The operations that make up a transaction act on what appears to be a private copy of the objects. This copy is called a **workspace**. It is only when the user tells the DBMS to **commit** the current transaction that it tries to merge the modified objects in the user's workspace with the main shared object store (i.e., the DBMS).

In general, when the user tells the DBMS to commit a transaction, two conditions are tested that would indicate conflict with the activities of other concurrent users:

First, it determines whether or not other concurrent sessions have committed transactions of their own, modifying objects that were accessed during this user's transaction. If so, then outdated values may have been used in some computations.

Secondly, it checks for locks (see section 3.1.2) set by other sessions that would indicate their intention to modify objects that this user has already read or to read objects that this user has already modified. The presence of such locks would mean that committing the changes might invalidate another user's work.

If neither of these conditions holds, then the transaction is committed. This not only makes any new and/or modified objects visible to others as a permanent part of the shared database, but also makes visible to this user any new/modified objects that have been committed by others.

If, on the other hand, the system finds a conflict, then it refuses to commit the modifications. When a transaction fails to commit, it leaves the user's workspace intact with all of the new and modified objects which it contains. The user can then abort the transaction and start a new one. This discards all of the new objects and modifications from the aborted transaction. Depending on the activities of other users, the user may then be able to repeat the operations using the new values from the database and commit the new transaction without encountering any conflicts.

It should be realized that the process of committing a transaction is an "all-or-nothing" method of posting the user's updates to the main object store. The system either commits all of the modifications encapsulated in a transaction at once, or it commits none of them. Because of this property, transactions are said to be *atomic* (i.e., treated as a single, indivisible operation by the computer). All permanent object modifications are encapsulated in transactions and are therefore atomic. The system moves from one internally consistent state to another as users commit their changes, and no inconsistent data can be introduced as a result of concurrent changes that conflict with one another. One can think of the entire set of operations encapsulated within a transaction as occurring in the instant when the transaction is committed.

3.1 CONCURRENCY CONTROL MECHANISMS

Concurrency control mechanisms can be categorized into two main approaches: they either **prevent** conflicting actions during transaction execution (*pessimistic concurrency control*); or they **discard** conflicting updates when a transaction attempts to commit (*optimistic concurrency control*) [BMOP89].

3.1.1 OPTIMISTIC CONCURRENCY CONTROL

In the optimistic approach, users simply read and write objects at will as if they were the only user. The system searches for and detects conflicts with other sessions only at the time that the user tries to commit the transaction. Although relatively easy to implement, this mechanism entails the risk that the user will lose all of the work that was done if conflicts are detected and the transaction is aborted.

3.1.2 PESSIMISTIC CONCURRENCY CONTROL

In the pessimistic mechanism, there are two main techniques: one based on locks, the other based on timestamps.

A **lock** is a variable associated with an object in the database which describes the status of that object with respect to the possible operations that can be applied to it. Generally, there is a separate lock for each object in the database. Locks are used as a means of synchronizing the access of concurrent transactions to the database.

Using this technique, users act as early as possible to detect and prevent conflicts by explicitly requesting locks which signal their intention to read or write objects. If a user succeeds in locking an object, then other users will be unable to use the object in a way that would conflict with the locking user's purposes. If a user is unable to acquire a lock, then someone else has already locked the object. Therefore the user requesting the lock cannot use the object and then commit. The user can then abort the transaction immediately, rather than wasting time on work that cannot be committed. [Serv89]

A **timestamp** is a unique identifier created by the DBMS to identify a transaction. Typically, timestamps are assigned when transactions are submitted to the DBMS, so a timestamp can be thought of as "transaction start time". Transactions can therefore be ordered according to their timestamps to ensure serializability [EN89].

Timestamps can be generated in several ways. One possibility is to have a counter that is incremented each time its value is assigned to a transaction. In this scheme, the transaction timestamps are numbered 1,2,3,... Since a computer counter has a finite maximum value, the system must periodically reset the counter to zero (possibly when no transactions are executing for some short period of time). Another way to implement timestamps which avoids this reset problem is to use the current value of the system clock. [EN89]

Since timestamps are automatically created by the DBMS, the user has little or no control over them. In the locking mechanism, however, it is up to the user whether or not to request a lock on an object. Thus the user is able to exert some control over concurrency when using this mechanism. Consequently, in further discussions of pessimistic concurrency control, emphasis will be given to locking mechanisms.

3.1.3 OPTIMISTIC vs PESSIMISTIC CONCURRENCY CONTROL

Optimistic concurrency control is the most efficient mode of operation if [Serv89]:

- The user is not sharing data with other sessions; or

- The user is only reading data (i.e., not writing); or

- The user is writing a limited amount of shared data and can tolerate not being able to commit work some of the time.

In the optimistic mode, the system only looks for conflicts at commit time. Therefore the probability of a user being in conflict with others increases with the amount of time between commits and with the size of the user's read set (i.e., with the number of objects that are read).

Controlling concurrent access with locks (pessimistic concurrency control) is the most efficient mode of operation if [Serv89]:

- There is a lot of competition for shared data in the user's application; or

- The user cannot tolerate even an occasional inability to commit.

It is important to keep in mind that the locking mechanism improves one user's chances of committing only at the expense of others. Thus, locks should be used sparingly to prevent an overall degradation of system performance.

3.2 OODBMS CONCURRENCY CONTROL

The EXODUS [CDRS89] and ORION systems both use a locking technique (pessimistic approach) for concurrency control. In particular, ORION provides a mechanism that allows the locking of a composite object along with all of its component objects as a single unit. The AVANCE object management system [BH89] uses the timestamp model (pessimistic approach) for concurrency control. The Vbase [Onto88], ONTOS, and GemStone systems all provide both pessimistic and optimistic concurrency control.

The default mechanism in GemStone is optimistic concurrency control. That is, this mechanism is always in effect for objects which the user has not explicitly locked. A transaction that fails to commit leaves the user's workspace intact with all of the new and modified objects which it contains. The user may then either take some action to save the values of those objects in a file outside of GemStone, or abort the transaction altogether. [Serv89]

In the pessimistic concurrency control mode, three kinds of locks are provided: read, write, and exclusive. A session may hold only one kind of lock on an object at a time.

Holding a **read lock** on an object means that computations can be made based on the object's value and then committed without fear that some other transaction might have committed a new value for that object after the read lock was obtained. Another way of saying this is that holding a read lock on an object guarantees that other sessions cannot:

- Acquire a write or exclusive lock on the object; or

- Commit if they have written the object.

Multiple sessions may hold read locks on the same object. Therefore, read locks are also known as shared locks.

Holding a **write lock** on an object guarantees that the user can write the object and commit. That is, it ensures that the user will not find that someone else has prevented them from committing by writing the object and committing it before them during the transaction. Alternatively, it can be said that holding a write lock on an object guarantees that other sessions cannot:

- Acquire any kind of lock on the object; or

- Commit if they have written the object.

Write locks differ from read locks in that only one session may hold a write lock on an object. In fact, if a session holds a write lock on an object, then no other session may hold any kind of lock on the object. This prevents other sessions from receiving the assurance implied by a read lock that the value of the object in its workspace will not be outdated when it attempts to commit. Other sessions may, however, still read the object without acquiring any type of lock of their own.

An **exclusive lock** is like a write lock in that it guarantees the user's ability to write an object. However, it goes beyond a write lock by guaranteeing that other sessions cannot:

Acquire any kind of lock on the object; or
Commit if they have read or written the object.

GemStone's exclusive locks correspond to what traditional DBMSs call exclusive locks or sometimes just write locks. By contrast, GemStone's write locks are not exclusive in the conventional sense, as other sessions may still be able to commit after reading a write-locked object optimistically (i.e., without holding a lock).

In addition to the locks for single objects, GemStone also allows locks on collections of objects.

3.3 CONCURRENCY CONTROL IN THE TACTICAL DATABASE

The TDB will interact with four external interfaces: the Radar System (sensors interface), the Link 11 System (data link interface), the Navigation System (navigation interface), and the user or LCCDS operator (man-machine interface) [dePa90, SS90]. Data will be written to the TDB through these four external interfaces, but only the user (through the man-machine interface) will be allowed to read data from the TDB.

Additionally, some updates to the TDB will occur internally. That is, the system will automatically read data from the TDB, perform various calculations (for example, dead reckoning or waypoint maneuvering geometries) and then write the results back to the TDB.

Ownship position and velocity, waypoint maneuvering geometries, and track data are updated at least every four seconds. Also, the system shall automatically perform dead reckoning computations for all tracks if no update is received within a four second period of time. [SS90]

To meet these requirements, all automatic operations performed by the system will have to be short duration transactions (at most four seconds). For short duration transactions, the optimistic concurrency control technique is the most efficient approach.

Therefore, it is recommended that optimistic concurrency control be used for all automatic updates (either Local Auto or Remote) performed by the Tracking, Data Link, and Navigation Systems. If a transaction (update) fails to commit, it is recommended that the transaction be aborted (discarding the data), because the next update, with new data, will occur within the next four seconds.

On the other hand, updates performed manually by the user (in the Local Manual mode) will most likely take longer than four seconds and will therefore require locks of some type. Otherwise they would have little chance to commit as the automatic updates would almost certainly commit first.

It is therefore recommended that pessimistic concurrency control be used for all updates performed by the user (Local Manual updates). Once the user acquires a lock on an object, the transaction is then guaranteed to commit.

Transactions involving schema modification (i.e., changes to the definition of a class) will also require pessimistic concurrency control. To avoid inconsistencies in such cases, it is recommended that the user obtain exclusive locks not only on the classes being modified, but also on all of their descendant classes.

4 CRASH RECOVERY

Transaction recovery is concerned with the preservation of the atomic property of transactions. This means that, ideally, despite all possible failures of the computer system, all updates of every committed transaction will be stored in the database, and no update of any aborted transaction will be stored in the database. Of course, no system can support transaction recovery against all possible failures, which may include simultaneous failures of processor(s), main memory, secondary memory, and communication medium between processors. Most commercial DBMSs support recovery from **soft crashes** (which leave the contents of the disk intact) and from **hard crashes** (which destroy the contents of a disk). [KBCG89]

4.1 OODBMS RECOVERY

In ORION, ONTOS, Vbase, and GemStone, as well as most multi-user systems, the unit of recovery from soft crashes is the transaction. This means that changes made by committed transactions are kept, and changes not yet committed are lost.

To guard against hard crashes, GemStone and Vbase allow the user to create backup copies of the database. However, in Vbase the backup copy of the database is immutable; it can be deleted or replaced, but not modified [Onto88]. In GemStone, on the other hand, replicates of the database can be created. The system copies all objects from the file that contains the database to a new file, and afterward stores newly-committed objects in both files. Subsequent damage to one file leaves objects in the other one intact, allowing the system to continue to function normally with no loss of data. Only authorized users are allowed to create database replicates. GemStone allows the creation of up to 6 replicates. Of course, maintaining replicates of the database on line has a cost in both time and space.

4.2 RECOVERY IN THE TACTICAL DATABASE

Soft crashes in the TDB could result in the loss of data not yet committed to the database. Since most transactions in the TDB are of short duration (automatic transactions are expected to take less than four seconds, while manual transactions, in general, take just a few minutes), only recent information will be lost. Therefore, unless the soft crash is caused by hardware failure (which could cause the system to be "down" for a considerable amount of time), the database can be rapidly updated with new information.

Since GemStone allows authorized users to create replicates of the database, at least one backup copy (replicate) of the database should be kept on disk (preferably on a separate disk). This will greatly reduce the chances of losing data due to hard crashes.

5 GARBAGE COLLECTION

Each time a transaction commits, memory locations containing the old version of any modified data become unusable (and usually inaccessible as well). These locations are then considered to be "garbage" as they are not part of free space but do not contain usable information either.

Periodically, it is necessary to find all the garbage locations and add them to the list of free space. This process is called **garbage**

collection, which of course imposes additional overhead and complexity on the system. [KS86]

5.1 GARBAGE COLLECTION IN THE TACTICAL DATABASE

In GemStone, garbage collection is executed in single-user mode and can only be performed by authorized users. Since all qualified LCCDS operators should be capable of fully operating the system, they should all be authorized to perform periodic garbage collection.

The frequency in which garbage collection should be performed will depend, in general, on the number of objects modified or deleted during LCCDS operation. Since garbage collection may be a relatively long process requiring that no interfaces be logged in to the database, it is recommended that it be performed only in situations of low risk to the ship or when absolutely necessary (e.g., if performance should become severely degraded due to lack of free space).

6 DATABASE SECURITY

Data stored in a database needs to be protected from unauthorized access, malicious destruction or alteration, and accidental introduction of inconsistency. The term **database security** usually refers to security from malicious access, while **database integrity** refers to the avoidance of accidental loss of consistency [KS86]. Accidental loss of consistency may result from [KS86]:

- Crashes during transaction processing.

- Anomalies due to concurrent access to the database.

- Anomalies due to the distribution of data over several computers.

- A logical error which violates the assumption that transactions preserve database consistency constraints.

The techniques of recovery and concurrency control are useful in protecting the database against accidental loss of consistency. It is generally easier to protect against loss of data consistency than to protect against malicious access, which includes the following [KS86]:

- Unauthorized reading of data (theft of information).

- Unauthorized modification of data.

- Unauthorized destruction of data.

Absolute protection from malicious abuse is not possible. However, the cost to the perpetrator can be made sufficiently high to deter most, if not all, attempts to access the data without proper authority [KS86].

Therefore, in order to protect the database, security measures must be taken at several levels [KS86]:

Physical: The site or sites containing the computer systems must be physically secured against armed or surreptitious entry of intruders.

Human: Authorization of users must be done carefully to reduce the chance of an authorized user giving access to an intruder in exchange for a bribe or other favors.

Operating system: No matter how secure the DBMS is, weaknesses in operating system security could serve as a means of unauthorized

access to the database. Since almost all DBMSs allow remote access through terminals or networks, software-level security within the operating system is as important as physical security.

DBMS: Various restrictions on data stored in the database are possible. For example, some users could be authorized to access only a limited portion of the database while others could be allowed to issue queries, but forbidden to make modifications. The DBMS is responsible for enforcing restrictions such as these.

It is generally worthwhile to devote a considerable effort to the preservation of the integrity and security of the database. Loss of data, whether via accident or fraud, may seriously impair the ability of a "corporation" to function or, in the case of the TDB, could result in severe or catastrophic consequences.

6.1 OODBMS SECURITY

The System Documentation Manuals of Vbase [Onto88] and ONTOS [Onto90] do not mention security mechanisms. This suggests that these systems do not provide any mechanisms to avoid malicious access to the database.

GemStone, however, provides the following kinds of security mechanisms to help control access to sensitive code and data: login authorization, name hiding, procedural protection, and nonprocedural protection (authorization and privileges). The user may choose to employ any or all of these mechanisms.

6.1.1 LOGIN AUTHORIZATION

Login authorization provides GemStone's first line of protection. "Logging in" is the process of establishing a logical connection with GemStone that permits further interaction. It is analogous to logging in to a timeshared operating system. The GemStone system administrator, or someone with equivalent privileges, establishes user ID's and passwords for authorized users. [Serv89]

6.1.2 NAME HIDING

The system administrator assigns to each user a symbol list, which contains the names of all the system-defined objects that the user will be allowed to access. Although the decision about which objects to include is entirely up to the system administrator, the user's symbol list typically contains [Serv89]:

- A system dictionary which contains some or all of the system-defined classes, and any other objects that all GemStone users have access to. Although users can read the objects in this dictionary, they are generally not permitted to modify them.

- A private dictionary for user-defined objects that are not to be shared with other users.

- One or more special-purpose dictionaries which can be shared.

It can be difficult, or even impossible, for users to refer to global objects that are not in their symbol list. Consequently, just omitting objects which are to be "off-limits" to a user from their symbol list provides a certain amount of security. However, determined users may still find ways to circumvent this, since it is difficult to ensure that all indirect paths to an object are eliminated. [Serv89]

6.1.3 PROCEDURAL PROTECTION

If the user's program accesses objects only via methods, the use of objects can be controlled by including user identity checks within their methods. Obviously, this kind of checking on a large scale would require a fair amount of code, and, consequently, might be troublesome to maintain and update. [Serv89]

6.1.4 NONPROCEDURAL PROTECTION

GemStone provides two global mechanisms which are nonprocedural: authorization and privileges [Serv89]. The **authorization** mechanism protects objects from access by unauthorized users (i.e., those users who have not been explicitly given permission to access the object). Every user can utilize the authorization mechanism to protect both data and code objects on a selective basis.

GemStone enables the user to endow both other users and objects with authorization attributes. Whenever a program tries to read or write an object, GemStone compares the object's authorization attributes with those of the user whose program is attempting to do the access. If the two share the appropriate attributes, then the operation may proceed. If not, GemStone returns an error notification.

The authorization mechanism also allows the user to authorize reading an object without giving authority to write it. Authorizations are easy to change, and no extra code is required in the objects being protected.

The **privilege** mechanism, which is entirely independent of the authorization system, enables the system administrator to control who can send certain powerful messages, such as halting the system or changing passwords. Privileges are associated with only a few such methods, and the mechanism cannot be extended to control other methods.

Thus, while authorization can be used to control access to any object, the privilege mechanism guards a small but crucial set of methods. The system administrator determines who shall be privileged to invoke each of a small group of messages which, together, exert "life-and-death" control over all of the objects and functions in GemStone.

6.2 SECURITY IN THE TACTICAL DATABASE

The TDB will be used in a multi-user form, being shared by the Track System, Link 11 System, Navigation System, and LCCDS operator (user). However, since all transactions performed by the Track System, Link 11 System, and Navigation System are automatic (i.e., following a pre-programmed scheme), the only manual access to the database will be performed by the LCCDS operator. Since all LCCDS operators should be capable of fully operating the system in all circumstances, there should be very little need to control access to sensitive code and data. Thus, all qualified LCCDS operators should be given "login authorization" as well as having access to all objects in the TDB (i.e., they should all have the privileges of a system administrator).

7 CONCLUSIONS

Clustering, concurrency control, crash recovery, garbage collection, and security all have a direct influence on data reliability and database

performance. Data reliability (integrity) is maintained, as much as possible, by means of the concurrency control, crash recovery, and security mechanisms. The overall performance of a database is greatly influenced by the clustering and garbage collection techniques provided by the OODBMS used.

REFERENCES

- [BH89] A. Bjornerstedt and C. Hulten. "Version Control in an Object-Oriented Architecture". In [KL89], pp 451-485.
- [BCGK87] J. Banerjee, H-T. Chou, J.F. Garza, W. Kim, D. Woelk, N. Ballou, and H-J. Kim. "Data Model Issues for Object-Oriented Applications", *ACM Transactions on Office Information Systems*, Vol 5, No 1, Jan 1987, pp 3-26.
- [BMOP89] R. Bretl, D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, E.H. Williams, and M. Williams. "The GemStone Data Management System". In [KL89], pp 283-308.
- [CDRS89] M.J. Carey, D.J. DeWitt, J.E. Richardson, E.J. Shekita. "Storage Management for Objects in EXODUS". In [KL89], pp 341-369.
- [dePa90] E.G. de Paula. *A Tactical Database for the Low Cost Combat Direction System*. Master's Thesis, Naval Postgraduate School, Monterey, CA, Dec 1990.
- [EN89] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings Publishing Co, Inc, Redwood City, CA, 1989.
- [HZ87] M.F. Hornick and S.B. Zdonik. "A Shared, Segmented Memory System for an Object-Oriented Database", *ACM Transactions on Office Information Systems*, Vol 5, No 1, Jan 1987, pp 70-95.
- [KBCG89] W. Kim, N. Ballou, H-T. Chou, J.F. Garza, and D. Woelk. "Features of the ORION Object-Oriented Database System". In [KL89], pp 251-282.
- [KL89] W. Kim and F.H. Lochovsky, eds. *Object-Oriented Concepts, Databases, and Applications*, ACM Press (Addison-Wesley Publishing Co), New York, NY, 1989.
- [KS86] M.F. Korth and A. Silberschatz. *Database System Concepts*. McGraw-Hill, Inc, San Francisco, CA, 1986.
- [Onto88] Ontologic Inc. *Vbase Integrated Object Database: System Documentation, Releases 0.8-1.0*. Ontologic Inc, Burlington, MA, 1988.
- [Onto90] Ontologic Inc. *ONTOS Object Database Documentation, Release 1.5*. Ontologic Inc, Burlington, MA, 1990.
- [Ross89] D.L. Ross. *Object-Oriented Database Manager for the Low Cost Combat Direction System*. Master's Thesis, Naval Postgraduate School, Monterey, CA, Dec 1989.
- [Serv89] Servio Logic Corp. *Programming in OPAL*. Servio Logic Corp, Beaverton, OR, 1989.
- [SS90] J. Seveney and G. Steinberg. *Requirements Analysis for a Low Cost Combat Direction System*. Master's Thesis, Naval Postgraduate School, Monterey, CA, Jun 1990.

DISTRIBUTION LIST

| | |
|--|----------|
| Defense Technical Information Center Cameron Station Alexandria, VA 22314 | 2 copies |
| Library, Code 0142 Naval Postgraduate School Monterey, CA 93943 | 2 copies |
| Center for Naval Analyses 4401 Ford Avenue Alexandria, VA 22302-0268 | 1 copy |
| Director of Research Administration Code 81 Naval Postgraduate School Monterey, CA 93943 | 1 copy |
| Maj M.L. Nelson, USAF Naval Postgraduate School Code CS, Dept. of Computer Science Monterey, CA 93943 | 5 copies |
| Professor LuQi Naval Postgraduate School Code CS, Dept. of Computer Science Monterey, CA 93943 | 5 copies |
| Capt E.G. de Paula, Brazilian AF Centro Técnico Aeroespacial (CTA) IAE - ESB São José dos Campos, SP, Brazil, 12225 | 5 copies |